Mechanized Reasoning About Actions Specified in $A^*$

Sakthi Subramanian,
Department of Computer Sciences,
University of Texas at Austin,
TX 78712.
email: sakthi@cs.utexas.edu

Abstract
In order to prove that a sequence of actions can transform an initial situation of the world to a goal situation when complete knowledge of the world is never available, default rules that serve to complete partial descriptions of the world are usually deemed necessary. This leads to non-monotonic reasoning which has proven to be difficult to formalize. In this paper, we present a mechanized formalization for default reasoning about actions specified in the language $A$ in the Boyer-Moore logic, a first-order logic. The main idea is to use the Boyer-Moore logic as a meta-language for formalizing the mapping from partial descriptions of the world to the models of the world usually formalized by applying default rules to partial descriptions. We formalize this mapping by including partial descriptions of the world as explicit objects in the universe of discourse and define a Lisp interpreter that takes a partial description as an argument and simulates the default behavior of the world by applying the "commonsense law of inertia" to the partial description. We formalize some typical examples and show mechanical proofs requiring default reasoning about actions.

Introduction
In order to prove that a sequence of actions can transform an initial situation of the world to a goal situation when complete knowledge of the world is never available, default rules that serve to complete partial descriptions of the world are usually deemed necessary. This leads to non-monotonic reasoning which has proven to be difficult to formalize. In this paper, we present a mechanized formalization for default reasoning about actions specified in the language $A$ in the Boyer-Moore logic, a first-order logic. The problem of formalizing default reasoning is usually taken to be the discovery and formal expression of default rules that can extend any partial description of the world to a complete description that formalizes a model of the world. We take a different approach and formalize default reasoning by using the Boyer-Moore logic as a meta-language for describing the mapping from syntactic partial descriptions of the world to the models of the world that are formalized by applying default rules to partial descriptions. We formalize this mapping by including partial descriptions of the world as explicit objects in the universe of discourse and define a Lisp interpreter that takes a partial description as an argument and simulates the default behavior of the world by applying the "commonsense law of inertia" to the partial description. We formalize some typical examples and show mechanical proofs requiring default reasoning about actions. The proofs were done completely automatically by the Boyer-Moore theorem prover.

The Language $A$
Gelfond and Lifschitz ([Gelfond and Lifschitz, 1992]) proposed a high-level language $A$ for specifying partial descriptions of the world and defined its formal semantics. In $A$, partial descriptions of the world are specified as domains. A domain consists of a set of fluent names, a set of action names and a set of propositions. There are two kinds of propositions in a domain: v-propositions and e-propositions. A v-proposition specifies the value of a fluent in a particular situation—either in the initial situation, or after performing a sequence of actions. An e-proposition describes the ef-
flect of an action on a fluent. A v-proposition is an expression of the form

\[ F \text{ after } A_1; \ldots; A_m, \] (1)

where \( F \) is a fluent expression (fluent name or its negation), and \( A_1; \ldots; A_m \) (\( m \geq 0 \)) are action names. If \( m = 0 \), we will write (1) as

initially \( F \).

An e-proposition is an expression of the form

\[ A \text{ causes } F \text{ if } P_1, \ldots, P_n, \] (2)

where \( A \) is an action name, and each of \( F, P_1, \ldots, P_n \) (\( n \geq 0 \)) is a fluent expression. About this proposition we say that it describes the effect of \( A \) on \( F \), and that \( P_1, \ldots, P_n \) are its preconditions. If \( n = 0 \), we will drop if and write simply

\[ A \text{ causes } F. \]

Example 1. The Fragile Object domain, motivated by an example from [Schubert, 1990], has the fluent names Holding, Fragile and Broken, and the action Drop. It consists of two e-propositions:

\[ \begin{align*}
\text{Drop causes } &\neg\text{Holding if Holding}, \\
\text{Drop causes } &\text{Broken if Holding, Fragile}.
\end{align*} \]

Example 2. The Yale Shooting domain, motivated by the example from [Hanks and McDermott, 1987], is defined as follows. The fluent names are Loaded and Alive; the action names are Load, Shoot and Wait. The domain is characterized by the prepositions

\[ \begin{align*}
\text{initially } &\neg\text{Loaded,} \\
\text{initially } &\text{Alive,} \\
\text{Load causes } &\text{Loaded,} \\
\text{Shoot causes } &\neg\text{Alive if Loaded,} \\
\text{Shoot causes } &\neg\text{Loaded.}
\end{align*} \]

To describe the semantics of \( A \), we will define what the "models" of a domain description are, and when a v-proposition is "entailed" by a domain description.

A state is a set of fluent names. Given a fluent name \( F \) and a state \( \sigma \), we say that \( F \) holds in \( \sigma \) if \( F \in \sigma \); \( \neg F \) holds in \( \sigma \) if \( F \notin \sigma \). A transition function is a mapping \( \Phi \) of the set of pairs \( (A, \sigma) \), where \( A \) is an action name and \( \sigma \) is a state, into the set of states. A structure is a pair \( (\sigma_0, \Phi) \), where \( \sigma_0 \) is a state (the initial state of the structure), and \( \Phi \) is a transition function. A v-proposition (1) is true in a structure \( M \) iff \( F \) holds in the state \( \Phi(A_m, \Phi(A_{m-1}, \ldots, \Phi(A_1, \sigma_0) \ldots)) \).

A structure \( (\sigma_0, \Phi) \) is a model of a domain description \( D \) if every v-proposition from \( D \) is true in \( (\sigma_0, \Phi) \), and, for every action name \( A \), every fluent name \( F \), and every state \( \sigma \), the following conditions are satisfied:

(i) if \( D \) includes an e-proposition describing the effect of \( A \) on \( F \) whose preconditions hold in \( \sigma \), then \( F \in \Phi(A, \sigma) \);

(ii) if \( D \) includes an e-proposition describing the effect of \( A \) on \( \neg F \) whose preconditions hold in \( \sigma \), then \( F \notin \Phi(A, \sigma) \);

(iii) if \( D \) does not include such e-propositions, then \( F \in \Phi(A, \sigma) \iff F \in \sigma \).

Item (iii) is the "commonsense law of inertia" ([Lifschitz, 1987, Gelfond et al., 1991]) according to which an action \( A \) leaves a value of a fluent \( F \) unchanged if there is no effect proposition describing the effect of \( A \) on \( F \).

The following example shows that the entailment relation of \( A \) is non-monotonic.

Example 3. Consider a Switch Domain with fluent names \( \text{Light1} \) and \( \text{Light2} \), an action \( \text{Switch1} \) and the following propositions.

\[ \begin{align*}
\text{initially } &\neg\text{Light1,} \\
\text{initially } &\neg\text{Light2,} \\
\text{Switch1 causes Light1.}
\end{align*} \]

It is clear that the Switch Domain entails

\[ \neg\text{Light1 after Switch1.} \] (3)

because there is no e-proposition that describes the effect of \( \text{Switch1} \) on \( \text{Light2} \). Therefore, it is assumed by default that \( \text{Switch1} \) leaves \( \text{Light2} \) unchanged. Suppose we extend the domain with the e-proposition

\[ \text{Switch1 causes Light2.} \] (4)

then (3) is not entailed by this Extended Switch Domain since it follows that \( \text{Switch1} \) makes \( \text{Light2} \) true when executed in any situation.

Thus, the entailment relation of \( A \) is non-monotonic: when an e-proposition is added to a domain \( D1 \) to form a new domain \( D2 \), the set of models of \( D2 \) is not in general a subset of the set of models of \( D1 \). Since the initial state is completely specified, both the Switch Domain and the Extended Switch Domain have exactly one model determined by the transition function specified by the domains. The addition of (4) to the Switch Domain modifies the transition function of the domain although the set of possible states and the initial state remain the same. On the other hand, the addition of a v-proposition to a domain does not result in non-monotonicity since it only eliminates those models of the domain in which it is false.

Our Approach

Our problem is to formalize the semantics of \( A \). First of all, we need a "translation" of \( A \), i.e., a formalization of the class of all possible models of all possible domains of \( A \). Such a formalization would allow us to "translate" particular domains as a set of axioms so that v-propositions entailed by a domain are provable. A translation of \( A \) can be accomplished in classical first-order logic but it is usually believed that "frame axioms" are needed ([Lin and Shoham, 1991,
Secondly, a formalization of $A$ must automatically apply default rules: if a domain $D_1$ is extended by adding e-propositions to obtain a domain $D_2$, our formalization must automatically switch to a translation of $D_2$ from a translation of $D_1$ when extended by a translation of the additional e-propositions. This is usually accomplished by default rules such as circumscription ([McCarthy, 1980, Lifschitz, 1987]).

The basic idea behind our approach is simple: if we include partial descriptions of the world as objects in the universe of discourse, then we can directly express the “commonsense law of inertia” in first-order logic by quantifying over descriptions. Thus, we can state directly in first-order logic that an action does not change the value of a fluent in a state unless explicitly stated in a description by quantifying over actions, fluents, truth values, states, and descriptions. For this, we must formalize the syntax of $A$, its semantics and the relationship between the two. We use the Boyer-Moore logic as a meta-language for formalizing the mapping from syntactic partial descriptions of the world to the models of the world that are usually formalized by completing partial descriptions using default rules. In the standard approaches, the mapping from the set of partial descriptions to the set of models of the world is not formalized. Rather it is used by the designers of default rules to prove its soundness and completeness. We formalize this mapping by including finite sets of effect propositions as explicit objects in the universe and define a Lisp interpreter in the Boyer-Moore logic ([Boyer and Moore, 1988]) that takes a list of actions $\ell$, a domain state $s$ and a set of effect propositions $E$ as arguments and returns the state got by executing $\ell$ in $s$ according to the transition function specified by $E$.

The “commonsense law of inertia” is asserted computationally using the Lisp interpreter: to compute the effects of an action, go through the list of effect propositions, determine the fluent expressions affected by the action and modify the given state accordingly. Since the interpreter simulates the default behavior of the world as a function of the set of e-propositions specified in a domain, formal reasoning about actions described by a domain can be done by reasoning about the interpreter. The value propositions of a domain may be used as hypotheses that constrain various states got by executing the interpreter on sequences of actions starting with an initial state.

This approach holds two advantages. Individual domains can be formalized without resorting to frame axioms. Default reasoning can be formalized because proofs about a domain do not “interfere” with proofs about an extension of the domain.

**Our Formalization**

We restrict ourselves to finite domains. We will describe our formalization using a conventional syntax rather than the official Lisp-like syntax of the Boyer-Moore logic.\(^1\) To formalize the semantics of $A$, we need to represent all possible states of finite domains, all possible sequences of actions of finite domains and all possible finite sets of e-propositions as terms in the language. We represent fluents and actions as litatoms.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fluentp $(z) = \text{litatom}(z)$</td>
<td>fluent expression $z$ is true in the state.</td>
</tr>
<tr>
<td>actionp $(x) = \text{litatom}(x)$</td>
<td>action expression $x$ is true in the state.</td>
</tr>
<tr>
<td>valuep $(x)$</td>
<td>if $x$ is either 0 or 1, the predicate valuep checks if its argument is either 0 or 1.</td>
</tr>
<tr>
<td>a-statep $(x) = \text{fvlistp}(x)$</td>
<td>a state is represented as a list of pairs of the form $\text{cons}(\text{fluent}, \text{value})$ where value is either 0 or 1. A fluent is true or false in a state depending on whether it is paired with 1 or 0.</td>
</tr>
</tbody>
</table>

**Theorem:** a-statep $(\text{STATE2})$

The interpreter is a function from syntactic descriptions of the world viz. sets of e-propositions to their semantic counterparts viz. the transition function that they define. To define the interpreter, we must formalize the syntax of $A$. We represent action names and fluent names by litatoms just like their semantic counterparts. A fluent expression is represented as a fluent-value pair. When interpreting e-propositions, we need to test if preconditions hold in a state. We define a predicate holds that takes a fluent expression and a state as arguments and checks if the fluent expression is true in the state. Since we allow a fluent to occur as the car of more than one fluent-value pair in a state,

\(^1\) The translation between the conventional syntax and the official Lisp-like syntax is discussed in [Boyer and Yu, 1992].
holds checks the first pair in the given state whose car is equal to the car of the given fluent expression using the built-in function assoc. Thus, \( \neg \text{Broken} \) holds in \text{STATE2} defined earlier.

**Definition:**
holds(fexp, s) = (assoc(car(fexp), s) = fexp)

**Theorem:** holds-example2
holds(' \text{(broken} . 0), \text{STATE2})

holds-all checks if all the fluent expressions in the given list \( x \) hold in state \( s \).

**Definition:**
holds-all (z, s) = ifz_\simnil then t
else (assoc (caar (x), s) = car (x)) \land
holds-all (cdr (z), s) endif

We represent an e-proposition as a list whose car is an action name, cadr is an effect which is a fluent expression and the rest are fluent expressions corresponding to preconditions. Thus, an e-proposition is a list whose car is an action name and whose cdr is a list of fluent-value pairs.

**Definition:**
e-prop (z) = (actionp (car (z)) \land fvlistp (cdr (z)))

For instance, the e-proposition:

*Drop causes Broken if Holding, Fragile.*

is represented as shown in the theorem below.

**Theorem:** eprop-example1
e-prop (' \text{(drop} (\text{broken} . 1) (\text{holding} . 1) (\text{fragile} . 1))

The predicate apgm recognizes a list of e-propositions.

**Definition:**
apgm(x) = if x \sim nil then t
else e-prop (car (x)) \land apgm (cdr (x)) endif

The example below shows the representation of the list of e-propositions in the Fragile Object domain.

**Definition:**
fo-domain = '('\text{(drop} (\text{holding} . 0) (\text{holding} . 1) (\text{broken} . 1) (\text{fragile} . 1))

**Theorem:** apgm-example1
apgm(fo-domain)

The interpreter resultlist takes a list of actions \( l \), a state \( s \) and a list of e-propositions \( pgm \) as arguments and returns the state got by executing \( l \) in \( s \) according to the transition function specified by \( pgm \). The interpretation of effect propositions is done by the function compute-effects. Compute-effects returns a list containing all fluent-value pairs made true by executing an action \( a \) in the given state \( s \) according to the set of effect propositions \( pgm \). The "commonsense law of inertia" is expressed using this function. The function goes through the list of effect propositions given by \( pgm \) and returns a list of all fluent expressions \( P \) such that there exists an effect proposition \( e \) describing the effect of \( a \) on \( P \) in \( pgm \) and the preconditions specified in \( e \) hold in \( s \).

**Definition:**
compute-effects (a, s, pgm) = if pgm \sim nil then nil
elseif (a = caar (pgm)) \land holds-all (cddar (pgm), s) then cons (cadar (pgm),
compute-effects (a, s, cdr (pgm)))
else compute-effects (a, s, cdr (pgm)) endif

Here is a theorem that shows how compute-effects operates. The list of effects of executing 'drop in \text{STATE2} includes \( \neg \text{Holding} \) and \text{Broken} since the preconditions in both the effect propositions of the Fragile Object Domain hold in \text{STATE2}.

**Theorem:** compute-effects3
compute-effects ('\text{drop}, \text{STATE2}, \text{FO-DOMAIN}) = '((\text{holding} . 0) (\text{broken} . 1))

The function result returns the state got by executing an action \( a \) in \( s \) according to the transition function specified by \( pgm \). The effects of the action in \( s \) computed by compute-effects are appended in front of \( s \) because we only look at the first occurrence of a fluent in a state to determine its truth value.

**Definition:**
result (a, s, pgm) = append (compute-effects (a, s, pgm), s)

**Theorem:** result-example3
result ('\text{drop}, \text{STATE2}, \text{FO-DOMAIN}) = '((\text{holding} . 0)
(broken . 1)
(\text{holding} . 1)
(\text{fragile} . 1)
(broken . 0)
(broken . 1))

Notice that we do not allow an inconsistent set of effect propositions to be given as an argument to result. We shall stick to consistent sets of effect propositions since conditions leading to inconsistent states can be treated as action preconditions ([Reiter, 1991, Gelfond et al., 1991]): an action may be prohibited from being executed in a state whenever it results in an inconsistent state.

The interpreter and an example of its output on concrete data are given below. When 'drop is executed once in \text{STATE2} it has the effect of making 'holding false and 'broken true. But when it is executed again it has no effect since the preconditions of all the effect propositions in the domain are false in the new state.
DEFINITION:
resultlist(l, s, pgm) = if l ≠ nil then s
else resultlist(cdr(l), result(car(l), s, pgm), pgm) endif

THEOREM: resultlist-example2
resultlist('((drop drop), STATE2, FO-DOMAIN) = '((holding . 0)
(broken . 1)
(holding . 1)
(fragile . 1)
(broken . 0)
(broken . 1))

Examples

We are now ready to formalize the example domains given in section 2 and prove theorems that follow from the domain descriptions. The theorems given below were proved completely automatically by the Boyer-Moore theorem prover.

From the Fragile Object Domain, it follows that if Holding, Fragile and ~Broken are true in a situation s0, then in the situation got by executing Drop in s0, ~Holding, Fragile and Broken would be true. The theorem can be expressed using the result function as shown below.

DEFINITION: holding(s) = holds('holding . 1), s
DEFINITION: fragile(s) = holds('fragile . 1), s
DEFINITION: broken(s) = holds('broken . 1), s

THEOREM: frag-th1
(a-statep(s0)
∧ holding(s0)
∧ fragile(s0)
∧ (~ broken(s0))
∧ (s1 = result('drop, s0, FO-DOMAIN)))
→ ( (~ holding(s1)) ∧ fragile(s1) ∧ broken(s1))

Notice that we have not restricted s0 to include only the three fluents Holding, Broken and Fragile. The above theorem holds in all of the infinite number of domains that can be got by adding new fluents to the Fragile Object Domain. Also, the state s0 is represented as a variable that ranges over the set of possible states rather than as a constant as in the usual situation calculus ([McCarthy and Hayes, 1969]) based formalisms.

Here is an example in which the initial state of the Fragile Object Domain is not fully defined. If Holding and ~Broken hold in s0 then ~Holding holds in the situation got by executing Drop in s0. Here we do not know if Fragile holds or not in the initial situation or in the resulting situation.

THEOREM: frag-th2
(a-statep(s0) ∧ holding(s0) ∧ (~ broken(s0)))
→ (~ holding(result('drop, s0, FO-DOMAIN))))

The set of effect propositions in the Yale Shooting Domain is defined below and is followed by the usual theorem about the death of Fred.

DEFINITION:
YALE-DOMAIN = '((load (loaded . 1))
(shoot (alive . 0) (loaded . 1))
(shoot (loaded . 0)))

THEOREM: ysp1
(holds('loaded . 0), s0) ∧
holds('alive . 1), s0))
→ holds('alive . 0), resultlist('((load wait shoot), s0, YALE-DOMAIN))

"Non-monotonic" reasoning can also be formalized in our theory. The theorem that Light2 remains false after Switch1 is executed in the Switch Domain is stated as follows.

DEFINITION:
SWITCH-DOM1 = '((switch1 (light1 . 1)))

THEOREM: l-th1
holds('light2 . 0), s)
→ holds('light2 . 0), result('switch1, s, SWITCH-DOM1)

If the Switch Domain is extended with the e-proposition:

Switch1 causes Light2

the theorem that Switch1 will make Light2 true can be proved about this Extended Switch Domain.

DEFINITION:
SWITCH-DOM2 = '((switch1 (light1 . 1))
(switch1 (light2 . 1)))

THEOREM: l-th2
holds('light2 . 0), s)
→ holds('light2 . 1), result('switch1, s, SWITCH-DOM2)

Acknowledgments

The idea of obtaining a "translation" of $A$ in the Boyer-Moore logic was suggested by Vladimir Lifschitz who also listened patiently to several half-baked formalizations. Bob Boyer gave me a hint that helped me make the jump from "translations" of $A$ to "modular translations". Hudson Turner, Norm McCain and G. N. Kartha also read a draft of the paper and offered suggestions. My many thanks to all of them.
References


